

Computer Lab 3: Multiple Server Queue with Reneging Customers

Concepts

- Simple use of containers
- Passing parameters in `waitDelay()` method
- Use of `interrupt()` to implement cancelling edge
- `CongruentialSeeds.SEED[]`
- Multiple Runs

Description

Impatient customers arrive to a multiple-server queue; each customer is only willing to wait a certain amount of time in the queue, after which he or she will “renege.” A reneging customer leaves the queue and never returns to the system. For the model, these “renege times” will be assumed to be independent identically distributed random variables, which will be denoted t_R . The Event Graph for the server portion of the model is shown in Figure 1.¹

The model in Figure 1 adds reneging by receiving unique customer objects upon arrival to the queue (i.e. at the Arrival event). This customer is added to the end of a fifo container (called ‘q’).² The Renege event is then scheduled, with the customer passed as a parameter. When the Renege event occurs, it removes the customer in its argument from the queue and increments the renege count (R). Whenever a StartService occurs first, however, the Renege event corresponding to that customer is cancelled.

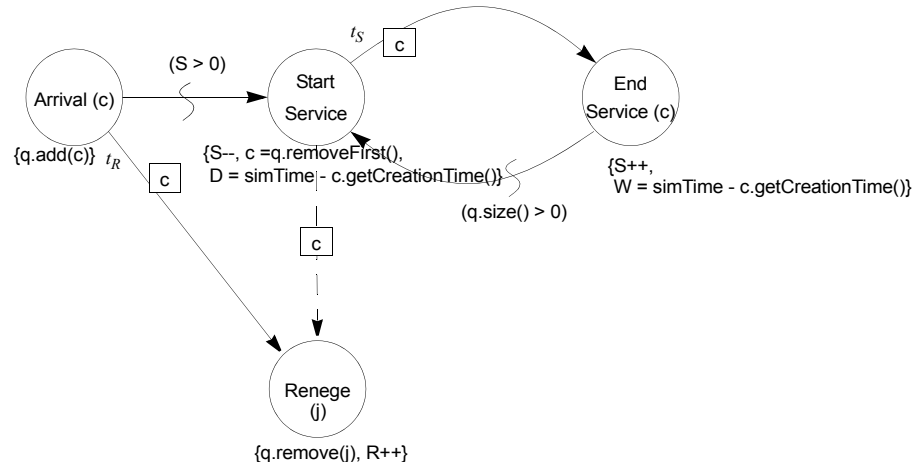


Figure 1. Event Graph for Multiple Server Queue with Reneging Customers

There are three new features of Simkit you will need to implement this model: defining events with arguments, passing parameters on edges, and cancelling edges. Additionally, you will be using some of the built-in random number seeds.

-
1. The arrivals will be generated using the `ArrivalProcess` class from Lab 01.
 2. The container `q` is a fifo queue with the additional property that elements can be removed from the middle as well.

The Customer Class

First define the `Customer` class (note that it does *not* subclass `SimEntityBase`). This will have two instance variables, one for the time the customer was “created”¹ and the second containing the renege time for the customer. This time should be passed into the `Customer`’s constructor as a double. These two instance variable should be exposed by (public) getter methods.

The CustomerCreator Class

Customers will be created by the `CustomerCreator` class. This is a `SimEntityBase` subclass that has a `RandomVariate` parameter to generate the renege times.² The Event Graph for `CustomerCreator` is shown in Figure 2

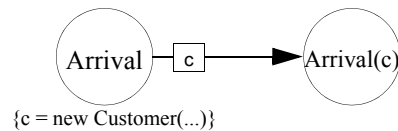


Figure 2. CustomerCreator Event Graph

The Event Graph in Figure 2 has the following implementation for the zero-parameter `Arrival` event:

```
public void doArrival() {  
    waitDelay("Arrival", 0.0, new Customer(renegeTime.generate())) ;  
}
```

The third argument to `waitDelay` passes the parameter to the event, as will be discussed next.

Defining Events with Arguments

Defining events with arguments is very easy in Simkit. Simply add the arguments to the corresponding “do” method. For the Event Graph in Figure 1 the `Renege` event has an argument which is a `Customer`. Thus, the `doRenege()` method in the `ServerWithReneges` class that implements Figure 1 should have signature `Customer`.³

Before writing the primary class, `ServerWithReneges`, here is some information about passing parameters on edges and implementing canceling edges in Simkit.

Passing Parameters on Edges

Parameters are passed as the third argument in the `waitDelay()` method corresponding to the scheduling edge. If there is only one element in the signature of the event being scheduled then it is simply added as a third argument.⁴ If the signature is a primitive type, then the third argument in the `waitDelay()` call should be the corresponding `Object` wrapper. For example, if the `Renege` event method is defined as `public void doRenege(Customer customer)`, then the `waitDelay()` call in the `doArrival(Customer)` method should be:

```
waitDelay("Renege", customer.getRenegeTime(), customer);
```

-
1. Use `Schedule.getSimTime()` to get the current value of simulated time.
 2. Recall that a parameter is implemented as a private instance variable with a setter and a getter.
 3. That is, `public void doRenege(Customer customer)`
 4. If there are more than one element in the signature, then they must be wrapped in an `Object[]` array.

where `renegeTime` is the randomly generated renege time and `customer` is a reference to the arriving customer.

Canceling Edges

Canceling edges are implemented in Simkit by invoking the `interrupt()` method (defined in `SimEntityBase`). The form you should use here has signature `(String, Object)`, where the first argument is the name of the event to be canceled and the second argument is the parameter corresponding to the event being cancelled. The first event that matches the *value* of the second parameter (as well as the name of the first parameter) of the `interrupt` will be removed from the event list. If there is no such event on the event list, then nothing happens.

In this case, the customer who is starting service must have his corresponding `Renege` event canceled, so the following code is used:

```
Customer customer = (Customer) queue.removeFirst();
...
interrupt("Renege", customer);
```

As with a scheduling edge, the call is made in the ‘do’ method corresponding to the event at the tail of the edge. Here, `queue` is a `LinkedList` containing the customer objects for all those customers in the queue who have not renege. The `interrupt()` statements should occur after the state transitions but before the `waitDelay()` statements.

The *ServerWithReneges* Class

The `ServerWithReneges` class processes customers according to the Event Graph in Figure 1. It has the same parameters as the `Server` class from Lab 02, but the state variables are slightly different, as shown in Table 1.¹

Table 1: Parameters and States for *ServerWithReneges* Class

Parameter	Type	State	Type
<code>numberServers</code>	<code>int</code>	<code>numberAvailableServers</code>	<code>int</code>
<code>serviceTime</code>	<code>RandomVariate</code>	<code>queue</code>	<code>LinkedList</code> ^a
		<code>numberServed</code>	<code>int</code>
		<code>numberReneges</code>	<code>int</code>

a. In the `java.util` package

Note carefully the ‘signatures’ for the events in Figure 1. The `Arrival(c)` event will be implemented by a `doArrival` method with signature `(Customer)`. After adding the incoming `Customer` to the queue, the `firePropertyChange` call looks like this:

```
firePropertyChange("numberInQueue", queue.size() - 1, queue.size());
```

Since the customer has just been added to the queue, the ‘old value’ of the number in the queue is one less than the current number. The `waitDelay()` that schedules the `Renege` event should get the renege time from the `Customer` and should pass the `Customer` instance as the third argument.

The `StartService` event has no argument, so the `doStartService()` method should likewise not either. Inside the method, you will need a reference to the `Customer` at the head of the queue, so use

1. Remember that parameters will have setters and getters, whereas state variables will only have getters.

the `removeFirst()` method of `LinkedList` to get the current customer. To implement the value of 'D' indicated in Figure 1, fire a `PropertyChange` event called "delayInQueue" whose value is the difference between the current time (`Schedule.getSimTime()`) and the time the current customer was created. After all states have been changed, invoke the `interrupt()` method, and finally the `waitDelay()` method.

Your constructor should have a signature the same as your `Server` class from Lab 02. Also, the `reset()` method should invoke `clear()` on the queue instance variable to empty it.

Execution Class

Your execution class should instantiate an `ArrivalProcess` instance, a `CustomerCreator` instance, and a `ServerWithReneges` instance. The `SimEventListener` structure is as shown in Figure 3.



Figure 3. SimEventListener Structure

The notation shows the `SimEventListener`s for the three objects. In Figure 3, the `CustomerCreator` instances is listening to the `ArrivalProcess` instance and the `ServerWithReneges` instance is listening to the `CustomerCreator` instance.¹

In addition to the two `SimpleStatsTimeVarying` instances for `numberInQueue` and `numberAvailableServers`, instantiate two instances of `SimpleStatsTally` with Strings "delayInQueue" and "timeInSystem" and add those two instances as `PropertyChangeListener`s to the `ServerWithReneges` instance. These use Tally statistics and will provide the estimates for the mean delay in queue and mean time in the system, respectively.

Parameters for Runs

Use the following parameters for your runs:

- Interarrival times are `Exponential(1.5)`
- Number of servers = 2
- Service times are `Gamma(2.5, 1.2)`
- Renege times are `Uniform(4.0, 6.0)`

There is a class in `simkit.random` called `CongruentialSeeds` that has a public static array of longs called `SEED`. These are 10 useful seeds that can be used to initialize `RandomVariate` instances. For this lab, use `CongruentialSeeds.SEED[0]` for arrivals, `CongruentialSeeds.SEED[1]` for service times, and `CongruentialSeeds.SEED[2]` for renege times.

Output

You should use verbose and/or single-step modes to debug your model. When you are satisfied that your model is working correctly, perform a run for 1000.0 time units, producing the following output:

1. Remember that for object `a` to be a `SimEventListener` to object `b`, the call is `b.addSimEventListener(a);`

```

Arrival Process
    Interarrival Times: Exponential (1.5)
Server with Reneging Customers
    Number Servers: 2
    Service Times: Gamma (2.5, 1.2)
    Reneging times: Uniform (4.0, 6.0)
Simulation ended at time 1000.000
    Number Arrivals: 665
    Number Served: 563
    Number Reneges: 96
    Percent Reneges: 0.1452
    Avg # in Queue: 1.5460
    Utilization: 0.8710
    Avg Delay in Queue: 1.9192
    Avg Time in System: 5.0070

```

The “utilization” is defined to be the average utilization per server.¹ The percent reneges should include those customers who have reneged, finished service, or are currently in service, but not those in the queue.

Deliverables

Turn in hard copies of your source code (except the `ArrivalProcess` class) and the output from the last long run (1000 time units).

Frequently Asked Questions

Can I have a primitive argument in my ‘do’ method?

Yes. However, the value must be wrapped in an Object when invoked in the `waitDelay()` statement. For example the method:

```

public void doThis(int j) {
    // do something
}

```

should be scheduled like this:

```

waitDelay("This", 1.0, new Integer(3));

```

What if my ‘do’ method has more than one parameter?

If a ‘do’ method has more than one parameter then you will also need to further wrap the Objects in an Object array. For example, the following method:

```

public void doThat(int k, double x, String s) {
}

```

should be scheduled using a call something like this:

```

waitDelay("That", 1.1,
    new Object[] {new Integer(42), new Double(3.141), "foobar"});

```

1. That is, $1.0 - \frac{\bar{s}}{k}$ where \bar{s} is the average number of available servers and k is the total number of servers.

Where do interrupt() calls go again?

Canceling edges are (by convention) executed after state transitions but before any scheduling edges. Therefore, `interrupt()` calls must be put in a 'do' method after all states have been changed and before any `waitDelay()` calls.